ASEE AMERICAN SOCIETY FOR ENGINEERING EDUCATION

Paper ID #44562

Towards Fuzz Testing a Procedurally-Generated Video Game

Dr. Erik Fredericks, Grand Valley State University

Erik Fredericks is an Assistant Professor in the School of Computing at Grand Valley State University. His research focuses on exploring how uncertainty can impact self-adaptive and safety-critical systems at different levels of abstraction and how it can be mitigated by using search-based software engineering techniques. Recently, he has been investigating how generative art can be automatically created via evolutionary computation.

Towards Fuzz Testing a Procedurally-Generated Video Game

Erik M. Fredericks
School of Computing
Grand Valley State University
Allendale, MI, 49401, USA
frederer@gvsu.edu

Skyler Burden
School of Computing
Grand Valley State University
Allendale, MI, 49401, USA
burdensk@mail.gvsu.edu

Abstract

Fuzz testing presents opportunities for discovering bugs in software projects that are unanticipated by developers as large amounts of either random or targeted inputs are applied to the system under test. Moreover, exploratory techniques such as search-based fuzz testing can discover new and interesting combinations of input data that can further lead to bug discovery. Video games are a subset of software projects that involve the additional overhead of audio/visual cues for gameplay, state management, and rigorous timing constraints. Procedural content generation (PCG) can be used to support development by incorporating unique game content (e.g., items, storylines, environments, etc.) via algorithms. As such, verification of PCG techniques is necessary to ensure that the generated content is valid for the situations in which they are deployed, given that such content can lead to emergent gameplay (i.e., unanticipated interactions that result in new features) or user dissatisfaction (e.g., the "same" type of rock is generated multiple times in a small area). We present our work-in-progress efforts and proposed run-time software testing methodology for developing an experimental testbed for fuzzing procedural generation in video games.

This project was created as part of the Grand Valley State University RISE Scholars program for first-generation students to participate in an active research program. `Delve the Dungeon` is our prototype framework for exploring how software engineering can enhance assurance that PCG techniques are executing as expected. Specifically, this framework provides a roguelike-style video game environment that comprises procedurally generated dungeons and text, with common features of this particular game domain including turn-based gameplay, bump-to-attack, and different forms of monsters that attack the player. Additionally, we have developed a proof-of-concept requirements specification to support our software engineering activities, where the next phase of development will monitor those requirements at run time, use the requirements as a basis for creating test cases and generating fuzzed test data, and then incorporate the results of run-time requirements monitoring and test case execution to the application as part of a feedback loop to continuously improve its behavior.

1. Introduction

Procedural content generation (PCG) is a technique for creating content using algorithmic approaches, most notably within video games.[4, 5] PCG has been widely used in recent years to significantly increase the amount of content within a game without needing to "hand-craft" each individual area and, moreover, provide a randomized yet cohesive experience each time the player starts a new game.[4, 5] PCG, however, generally requires inherent randomness to ensure that its algorithms yield different outputs. Balancing randomness with rigorous software constraints to ensure that the application continues to deliver a satisfactory level of quality is a non-trivial problem.

One approach in the field of software engineering for demonstrating quality is in software testing, or exposing software to inputs and/or situations and monitoring its response. While there are many different strategies for performing testing at multiple levels of abstraction,[1] we focus on fuzz testing with the goal of hardening a system against latent bugs and/or unexpected conditions.[8] More specifically, fuzz testing is a technique for generating a large amount of either randomized or guided test case data that may not have been considered when initially validating an application. For example, a fuzz test applied to a character controller within a video game might aim to simulate a large number of combinations of keyboard and mouse events (both valid and invalid) and then monitor if the game's response handles those events as expected. Our intended framework is planned to be executed at design time as well as at run time within the domain of video games that leverage PCG techniques.

This project was an undergraduate research project supported by an NSF-sponsored project for Retaining and Inspiring students in Science and Engineering (RISE) and yielded a work-in-progress video game framework for exploring fuzz testing. The rest of this paper is structured as follows. Section 2 discusses `Delve the Dungeon`, our research platform for roguelike games research, including relevant background information and related work. Section 3 then discusses our approach for student training and Section 4 summarizes our efforts and presents future directions for this path of research.

2. `Delve the Dungeon` - Roguelike Research Platform for Fuzz Testing

This section presents our work-in-progress experimental framework, `Delve the Dungeon`. We describe the application itself, discuss fuzz testing and its application to the video game domain, discuss our initial requirements and validation methods for this project, and present the next steps for this project within each section.

a. Overview of Application

`Delve the Dungeon` is a roguelike game that follows the Python `tcod` library tutorial series on roguelike development.[6] This tutorial is often used for introducing developers to roguelike game design practices. Additionally, the tutorial serves to demonstrate new Python programming practices (e.g., including data types, using NumPy for data manipulation, etc.). As such, this tutorial series served to onboard the undergraduate researcher into roguelike game development.

The base output of the tutorial includes common roguelike mechanics (e.g., bump-to-attack, field of view, tile-based/turn-based movement, etc.), as well as one form of map generation (i.e., generating random rooms and connecting them via tunnels). Figure 1a presents our implementation of the "main menu" screen and Figure 1b presents our implementation of the game interface. Our extensions to the tutorial include a quest for the player to follow, additional enemies and items, additional PCG algorithms for map generation, and a storyline that comprises procedurally-generated elements. The map in Figure 1b was generated using a cellular automata algorithm that typically is used for creating caverns.[5] Additionally, we included a PCG technique that uses a Simplex noise function for generating "smooth" features within the world.[5] For presentation purposes we do not go into detail for each of these algorithms.

Figure 1a: Delve the Dungeon Welcome Screen.

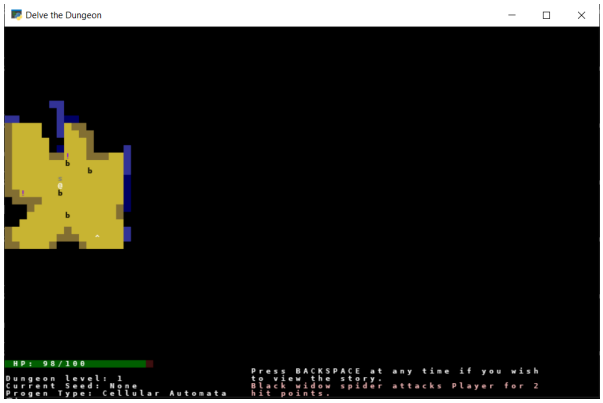

Figure 1b: Delve the Dungeon Game Screen.

Figure 1: Delve the Dungeon Screenshots.

Figure 2 presents a screenshot of the storyline synopsis presented to the user, comprising both hand-written and procedurally-generated elements.

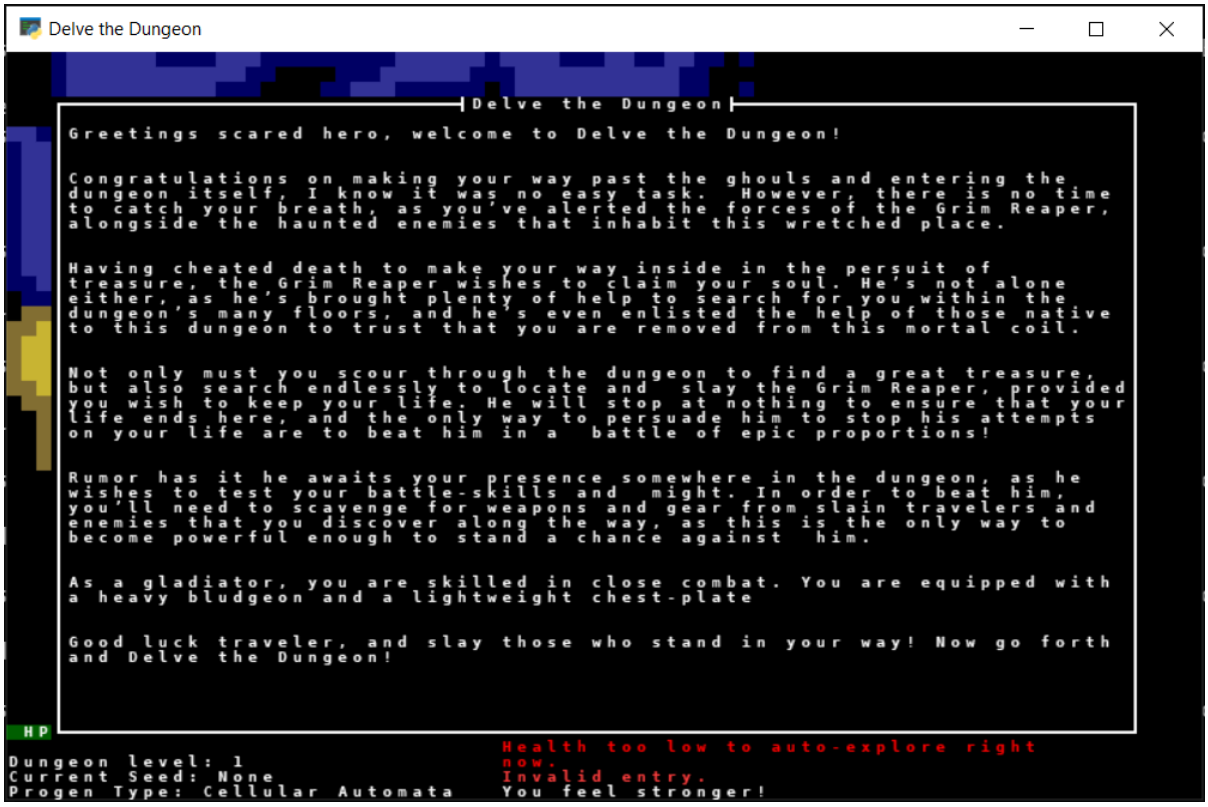

Figure 2: Delve the Dungeon Story with Procedurally-Generated Elements.

We used the Tracery library to enable story-based PCG, where this library uses an input grammar to specify possible options for prose output.[3] Listing 1 presents a sample of the rules that govern creation of the story synopsis:

```
synopsis_rules = {
    "player_adjective": ["brave", "scared", "terrified",
                         "strong", "tough", "capable"],
    "player_description": ["hero", "adventurer", "warrior",
                         "misfit", "traveler"],
    "entrance_type": ["undead army", "pit of lava", "main gate",
                         "treacherous drawbridge",
                         "royal guards", "ghouls"],
     "enemy_descriptor": ["various", "ghastly", "enraged",
                         "haunted", "powerful", "magical",
                         "horde of"],
      ...
}
```

Listing 1: Snippet of Tracery rules for creating story synopsis.

The rules in Listing 1 provide different options for Tracery to select when the synopsis is created. This grammar would be "flattened" (i.e., the grammar would be processed and random options selected by the library), yielding a string output. For example, any of the associated options for the keyword `player_adjective` could appear in the final output. Figure 2 shows the story synopsis screen, where the Tracery-generated output is inserted. In this particular output, the term `ghouls` appears to provide "flavor text" for the player. Running the program again would result in a different output, potentially yielding a different experience for the player.

b. Fuzz Testing

Fuzz testing is a strategy for finding latent flaws and/or bugs within software systems by subjecting them to an overwhelming number of random or guided test cases.[8] Fuzzing has been previously deployed as purely random, stochastic, grammar-based, and search-based, among others. Fuzzing has the downside, however, of resulting in a large number of superfluous test cases that may waste the tester's time (e.g., a large number of test cases within the same equivalence class).[2] This problem may be highly-noticeable within the domain of video game testing as many tests need to be executed manually (i.e., by the tester). As such, we aim to develop a fuzzer that is best suited for roguelike games and can be automatically executed. A side benefit of our domain is that roguelikes often are presented as purely ASCII games within terminal environments, leading to the possibility of reducing testing effort (e.g., testing values within a 2-D array is significantly easier than performing image processing on a rendered scene to ensure that your character is at the correct location).

Our aim for our future fuzz testing is to initially focus on the core implementation of our game as well as to ensure that the PCG elements are correct. For example, consider the activity of moving the player (`@`) around a 2D environment comprising walls (`#`) and floor (`.`). A sample space might look as follows in Listing 2:

```
##########
#....@...#
#........#
#..#..#..#
##########
```

Listing 2: Sample roguelike ASCII environment.

In this example, the player has five valid moves: left, right, down, down-left, and down-right. Moving in any of the other directions would be invalid as a wall blocks the player's path. Assuming we wanted to test if a move is valid, we could derive a fuzzer that creates sample

inputs to that function. For example, a function `bool isMoveValid(next_position: vec2)` expects a 2D vector as input and a Boolean as output. A fuzzer in this case would generate a large number of inputs to exercise the function that are both valid (i.e., a `vec2` within the expected range) and invalid (e.g., a `float`). We would expect that the fuzzed inputs cover all possible valid moves and that any invalid inputs are correctly handled (e.g., throw an `AssertionError` that a `float` is invalid input or that a `vec2` specifying characters as its values are also invalid).

We intend to develop numerous fuzzers for each aspect of our application in future extensions, from handling gameplay logic to ensuring that the PCG algorithms (i.e., generating storylines, creating and placing items and enemies, etc.) are also yielding correct values. Additionally, we will incorporate guided fuzzers to minimize the number of superfluous test cases.[2] For example, a grammar-based fuzzer would use a Backus-Naur notation to specify the expected configuration of a procedurally-generated item description.[8] Test cases created via this fuzzer would be constrained to that input grammar. Such an approach can minimize non-useful tests (e.g., an item description most likely would never be purely a `float` data type instead of a `string` datatype).

c. Software Requirements

One goal of this project was to derive an initial set of software requirements that specifies the intended behavior of our application. The intent of this document was twofold: to ensure that the final game meets our expectations and to provide a basis for deriving test cases to be used in the fuzz testing framework. As this project is a work-in-progress the requirements specification is in an early state, however the initial specification includes high-level requirements that are important to the success of our application. Additionally, the student provided a method for validating each requirement to be used as the basis for test case derivation.

For brevity we provide the full list in our GitHub repository (c.f., Section 3). As an example, Requirement 2 states that "`Items are placed only within the playable area`," and the validating method states that "`A flood fill algorithm must reach all items from the player's position within the map`." This requirement ensures that randomly-placed items are accessible to the player (e.g., they are not "stuck" within a wall) and provides one method of ensuring that the PCG algorithm for placing items is functioning correctly. Additionally, the validation method describes one possible approach for ensuring that the requirement is satisfied during execution. For reference, a flood fill algorithm aims to fill a space, emanating from a starting point (e.g., filling a shape with a color). This algorithm can also be used for reachability analysis in validating PCG algorithms.[7]

3. Student Training and Project Artifacts

This project was developed as part of an undergraduate research effort funded by the RISE Scholars program, where this program is intended to support students from impacted populations to provide opportunities within STEM disciplines. As such, our collective goals were to provide a funded research experience over the summer that would expose the student to new programming concepts, the field of software engineering, and the open-endedness of research. Each week we met to plan upcoming tasks, discuss progress and issues encountered, and expand upon our overall goals for the project.

Our main goal was to create an experimental framework for exploring software fuzz testing within the domain of roguelike video games, as well as to lay the groundwork for future empirical investigations. During the project's development we discovered that incorporating additional PCG elements and "polishing" the game would result in a better framework for future projects. In upcoming semesters we plan to incorporate a fuzz testing framework based on The Fuzzing Book,[8] an online resource for training students and practitioners in various forms of fuzz testing. In addition to coding, the student gained experience in the non-trivial task of deriving requirements for a project. While these requirements are still at an initial stage, they lay the groundwork for future expansion and derivation of test cases for verification and validation activities.

This project was developed as open source and is available in our GitHub repository, including our initial requirements specification: https://github.com/efredericks/ASEE-NCS-RoguelikeResearch. Our requirements file may be found in `sofware_requirements.md` within our repository.

Threats to Validity. This paper presents a discussion of our proof of concept application for performing testing on video games with procedural content. As such, we have identified the following threats to validity for this work. First, our testing framework has not been fully completed and as such may require significant efforts to include within `Delve the Dungeon`. Second, fuzz testing PCG-driven applications may be a non-trivial problem and also require significant effort to minimize the number of useless (i.e., requiring time and effort for no useful result) or flaky (i.e., producing inconsistent test results) tests, given the randomness inherent with PCG. We anticipate exploring both of these threats to validity in our follow-up projects. Our third threat is that fuzz testing can be problematic for incorporating within a video game environment (i.e., including PCG algorithms, visual outputs, user inputs, artificial intelligence for game entities, etc.). This last threat mainly focuses on the amount of time and effort required for future empirical study and may be considered positive in that there will be many problems to solve in the future.

4. Conclusion

This paper has presented our efforts and experiences in creating an open source experimental research framework for performing fuzz testing in roguelike video games, including development of an initial requirements specification to support testing. The project we have developed will support future empirical investigations in software testing, PCG, and software engineering topics relating to video games. Additionally, this project has supported the training of an undergraduate researcher in these fields.

Future work. We provided an overview of our work-in-progress experimental framework for exploring research topics in fuzz testing. Given its current status, our most pressing direction for future work is to finalize an empirical investigation into fuzzing within our environment. We aim to incorporate different types of fuzzers that impact PCG elements of the game (i.e., story generation, content generation), the interface, and the overall gameplay itself. Our goal is to discover potential problems within our environment while minimizing the amount of required test cases generated. Another direction for research is in search-based testing, or using evolutionary computation-based techniques for generating test cases. Last, we intend to continue developing the game itself for use in outreach activities and further research endeavors.

Acknowledgements

Financial support for this project was provided by RISE (www.gvsu.edu/rise), which is funded by a National Science Foundation S-STEM award No. 1742463 as well as Grand Valley State University. The views and conclusions contained herein are those of the authors and do not necessarily represent the opinions of the sponsors.

References

[1] Bertolino, A. "Software testing research: Achievements, challenges, dreams." In Future of Software Engineering (FOSE'07), pp. 85-103. IEEE, 2007.

[2] Brüning, F., Gleirscher, M., Huang W.L, Krafczyk, N., Peleska, J., and Sachtleben, R. 2023, September. Complete Property-Oriented Module Testing. In IFIP International Conference on Testing Software and Systems (pp. 183-201). Cham: Springer Nature Switzerland.

[3] Compton, K., Kybartas, B., and Mateas, M. "Tracery: an author-focused generative text tool," in International Conference on Interactive Digital Storytelling. Springer, 2015, pp. 154–161.

[4] Mason, S., Stagg, C., and Wardrip-Fruin, N. "Lume: a system for procedural story generation," in Proceedings of the 14th International Conference on the Foundations of Digital Games, 2019, pp. 1–9.

[5] Patel, A. "Making maps with noise functions." [Online]. Available: https://www.redblobgames.com/maps/terrain-from-noise/. Accessed 22 January 2024.

[6] Standridge, T. "Yet Another Roguelike Tutorial - Written in Python 3 and TCOD." *Roguelike Tutorials*. [Online]. https://www.rogueliketutorials.com/tutorials/tcod/v2/. Accessed 22 January 2024.

[7] Togelius, J., Champandard, A. J., Lanzi, P. L., Mateas, M., Paiva, A., Preuss, M., and Stanley, K. O. Procedural Content Generation: Goals, Challenges and Actionable Steps. In Artificial and Computational Intelligence in Games. Dagstuhl Follow-Ups, Volume 6, pp. 61-75, Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2013)

[8] Zeller, A., Gopinath, R., Böhme, M., Fraser, G., and Holler, C. "The Fuzzing Book." CISPA Helmholtz Center for Information Security, 2023. Retrieved 2023-01-07.